XR on Linux

User mode device driver

Modification History

Revision	Date	Originator	Comments
0.1	02/09/12	Etienne Martineau	

1 Introduction

User mode device drivers are required because:

- restart ability
- GPL isolation
- System stability
- Custom API (other than file operation)

2 Requirements:

2.1 Kernel mode driver should continue to work as expected.

2.2 Front-end / Back-end (NG)

2.3 Latency should be keep as minimal as possible

With fair share scheduling, if the run -queue become long, latency will grow. For predective latency we should have the user driver part of the real time classes A straightforward approach to improve responsiveness is to raise the priorities of threads and lift them into a real-time scheduling class. This comes with the risk of starving host services that the guest may indirectly busywait upon.

> CONFIG-PREEMPT-RT:
>
> spinlock_t
> Critical sections are preemptible. The _irq operations (e.g., spin_lock_irqsave()) do -not- disable hardware interrupts. Priority inheritance is used to prevent priority inversion. An underlying rt_mutex is used to implement spinlock_t in PREEMPT_RT (as well as to implement rwlock_t, struct semaphore, and struct rw_semaphore).
>
> raw_spinlock_t
> Special variant of spinlock_t that offers the traditional behavior, so that critical sections are non-preemptible and _irq operations really disable hardware interrupts. Note that you should use the normal primitives (e.g., spin_lock()) on raw_spinlock_t. That said, you shouldn't be using raw_spinlock_t -at- -all- except deep within architecture-specific code or low-level scheduling and synchronization primitives. Misuse of raw_spinlock_t will destroy the realtime aspects of PREEMPT_RT. You have been warned.

2.4 Hardware devices are owned by application and cannot be shared

Each device should have only a single driver. Therefore one needs a way to associate a driver with a device, and to remove that association automatically when the driver dies. This has to be implemented in the kernel, as it is only the kernel that can be relied upon to clean up after a failed process.

The simplest way to keep the association and to clean it up in Linux is to implement a new filesystem. Open files are automatically closed when a process dies, so cleanup also happens automatically. All files are closed when a process dies, so if there is a bug in the driver that causes it to crash, the system recovers ready for the driver to be restarted.

3 Peripheral IO memory

3.1 Resource carving

PCI Config MMIO PIO Interrupts Address Translation (DMA) Security Hardware ROMs

4 Programming model

Kernel space driver implement file operation; LD_PRELOAD provide such support.

4.1 Distributed devctl

5 Direct Memory Access DMA

5.1 IOMMU

Many modern architectures have an IO memory management unit, to convert from physical to I/O bus addresses -- in much the same way that the MMU converts virtual to physical addresses. The MMU also protects one virtual address space from another. Unfortunately, the IO MMU cannot totally visualized the PCI bus addresses. It can be set up to prevent DMA via bus addresses to arbitrary locations, but all mappings are seen by all PCI devices. On such systems, after the memory has been pinned, the IOMMU has to be set up to translate from bus to physical addresses; and then after the DMA is complete, the translation can be removed from the IOMMU.

5.2 DMA Types

The PCI code distinguishes between two types of DMA mappings, depending on how long the DMA buffer is expected to stay around:

5.2.1 Consistent DMA mappings

These exist for the life of the driver. Usually mapped at driver initialization, unmapped at the end and for which the hardware should guarantee that the device and the CPU can access the data in parallel and will see updates made by each other without any explicit software flushing and without any caching effect like burst for example.
Example: Network card DMA ring descriptors. The invariant these examples all require is that any CPU store to memory is immediately visible to the device, and vice versa. Consistent mappings guarantee this.

Under Linux, IO space is always mapped consistent.

Consistent DMA mapping are allocated / deallocated out of kmalloc

5.2.2 Streaming DMA mappings

Where, once the device has either read or written the area, it has no further immediate use for it. These are set up for a single operation. Some architectures allow for significant optimizations when streaming mappings are used, as we will see, but these mappings also are subject to a stricter set of rules in how they may be accessed. The kernel developers recommend the use of streaming mappings over consistent mappings whenever possible. There are two reasons for this recommendation. The first is that, on systems that support them, each DMA mapping uses one or more mapping registers on the bus. Consistent mappings, which have a long lifetime, can monopolize these registers for a long time, even when they are not being used. The other reason is that, on some hardware, streaming mappings can be optimized in ways that are not available to consistent mappings.

Streaming DMA mapping are build around existing memory which may or may not be discontiguous.

Also, streaming DMA mapping takes cares of processor's cache flushing at the moment the API is called.

Streaming DMA also automatically takes cares of address range addressability by the devices by setting up either bounce buffer OR the IOMMU.

5.3 Implementation options

5.3.1 Static memory carving

5.3.2 Dynamic API

5.3.3 Memory Allocators

This requirement push for a DMA management interfaces; 2 options:

1-Provide an allocators like what xrlinux has (packman / buffman) model

Require lwm / mmap(phys) / mmap_peer

NG ???

```
Posix typed memory
PHYS/Anon
QNX shm_control attribute (contiguous)
ftruncate Size ->DMA able
```

6 User space level Interruption

6.1 Interruption types

Legacy interrupts:

- Level-sensitive interrupts: These interrupts are generated as long as the physical interrupt signal is high. If the interrupt signal is not lowered by the end of the interrupt handling in the kernel, the operating system will call the kernel interrupt handler repeatedly causing the host platform to hang. To prevent such a situation, the interrupt must be acknowledged (cleared) by the kernel interrupt handler immediately when it is received. Legacy PCI interrupts are level sensitive.
- Edge-triggered interrupts: These are interrupts that are generated once, when the physical interrupt signal goes from low to high. Therefore, exactly one interrupt is generated. No special action is required in order to acknowledge this type of interrupt. ISA/EISA interrupts are edge triggered.

Message-Signaled Interrupts (MSI):

Newer PCI bus technologies, available beginning with v2.2 of the PCI bus and in PCI Express, support Message-Signaled Interrupts (MSI). This method uses "in-band" messages instead of pins and can target addresses in the host bridge. A PCI function can request up to 32 MSI messages.
Note: MSI and MSI-X are edge triggered and do not require acknowledgement in the kernel.

Among the advantages of MSIs:

- MSIs can send data along with the interrupt message.
- As opposed to legacy PCI interrupts, MSIs are not shared, i.e. an MSI that is assigned to a device is guaranteed to be unique within the system.

Extended Message-Signaled Interrupts (MSI-X) are available beginning with version 3.0 of the PCI bus. This method provides an enhanced version of the MSI mechanism, which includes the following advantages:

- Supports 2,048 messages instead of 32 messages supported by the standard MSI.
- Supports independent message address and message data for each message.
- Supports per-message masking.
- Enables more flexibility when software allocates fewer vectors than hardware requests. The software can reuse the same MSI-X address and data in multiple MSI-X slots.

The newer PCI buses, which support MSI/MSI-X, maintain software compatibility with the legacy line-based interrupts mecahnism by emulating legacy interrupts through in-band mechanisms. These emulated interrupts are treated as legacy interrupts by the host operating system

6.2 Interruption affinity

/proc/irq/IRQ#/smp_affinity specifies which target CPUs are permitted for a given IRQ source. t's a bitmask of allowed CPUs. It's not allowed to turn off all CPUs, and if an IRQ controller does not support IRQ affinity then the value will not change from the default 0xffffffff.

6.3 Interruption sharing

Whenever two or more drivers are sharing an interrupt line and the hardware interrupts the processor on that line, the kernel invokes every handler registered for that interrupt, passing each its own `dev_id`. Therefore, a shared handler must be able to recognize its own interrupts, and should quickly exit when its own device has not interrupted.

6.4 Linux low level Interrupt handling logic

Hardware device IRQ lines connect to input pins of interrupt controller (PIC).

- PIC monitors IRQ lines for raised signals.
- PIC converts signal to vector & stores it in an I/O port for CPU to access via data bus.
- PIC sends signal to INTR pin of CPU.
- <u>CPU enter into Interruption context with interruption disabled</u>
- CPU goes into do_irq; Mask and Ack the IRQ in the PIC
- PIC clears INTR line upon receipt of Ack from CPU.
    - From here PIC can issue other Interruption to CPU
- <u>CPU re-enable interruption</u> and grab a spin-lock for the given IRQ (to prevent other CPU from handling the same vector)
    - Only 1 CPU at a time into the handler
- CPU test and spin on the global interrupt lock (if set by the cli() instruction)
- CPU invoke handler
- --
- **(A)** Handler clear the devices IRQ line
    - Process critical information; pass it up to work-queue
- **(B)** Handler return IRQ_HANDLED
- --
- **(C)** CPU release spin-lock; CPU unmask PIC for the given vector
- **(D)** CPU leave Interrupt context

6.5 Implementation option

The Linux model requires step A and B to execute at interruption context; Without it, an interruption storm is going to be generated because of step C and D re-enabling the vector with devices hardwares lines left in asserted state.

The QNX model allows step A and B to be scheduled at user context because the implementation of step C and D are different.

When handling interruption at user space there is no need to use the following API for the sake of protecting global data:

- InterruptDisable
- InterruptEnable
- InterruptLock
- InterruptUnLock

Instead one could use regular mutex object to provide global data locking. There is a few cases where we may need Irq disable and Spinlock support:

-Critical time bound operation (cannot be scheduled out)

-Multi-thread / SMP system waiting for data coming out of Interruption context.

6.5.1 Option #1; Kernel devices specific handler

PROS:

- Acknowledge cycle logic for the PIC is left untouched.
- Fast and deterministic implementation
- Conform with Linux philosophy
- No modification to kernel code; Only device driver.
- SMP safe
- Interruption sharing is possible
- No priority based scheduling required

CONS:

- Require work-queue to avoid losing interrupt context
- Potentially more code is required on the driver side
- Require implementation of “shared memory queue” between user and kernel

6.5.2 Option #2; 100% user space handler

PROS:

- This is the current QNX model; { Actual implementation for the Demo on CRS/RP with ethserver sending/receiving packet }

CONS:

- Interruption sharing is impossible between kernel driver and user space driver; We cannot trust a user-space handler to return in a timely fashion therefore delaying kernel interruption delivery for other legitimate devices

- Require few modification for the PPC OpenPIC controllers code
- Require in depth modification of the X86 APIC controllers code because of it's inherent complexity on SMP
- Modification are directly in kernel code; Cannot be a device driver.
- Require new system call to control the IRQ disable / IRQ enable path
- SMI is another complex cases.

6.5.3 Option #3; 100% user space handler with Interruption controller masked

PROS:

- Compatible with the current QNX model;
- Acknowledge cycle logic for the PIC is left untouched.
- No modification to kernel code; Only device driver.
- SMP safe

CONS:

- Untested area of interruption cycle?
- Handler must run in the real time class.
- Require custom mechanisms to re-enable interruption (system call?)
- Interruption sharing is impossible between kernel driver and user space driver; We cannot trust a user-space handler to return in a timely fashion therefore delaying kernel interruption delivery for other legitimate device